Name: Andrew Hayes
Student ID: 21321503
Programme: 4BCT

CT4101

2024–10–31

Assignment 1: Classification Using Scikit-Learn

1 Description of Algorithms

1.1 Algorithm 1: Random Forest

Random decision forest is a supervised machine learning algorithm that can be used for both classification & regression that builds upon the **decision tree** algorithm by combining several decision trees to generate labels for a dataset. An implementation of this algorithm for classification is provided in scikit-learn as `sklearn.ensemble.RandomForestClassifier`[4]. While it can be used for regression as well as classification, I will only be referring to its use as a classification algorithm in this assignment, as regression is not relevant to the wildfire classification task at hand.

Since the random decision forest algorithm builds upon the decision tree algorithm, it is first necessary to explain briefly what decision trees are and how they work. A decision tree can be thought of a series of internal nodes (i.e., nodes which are not leaf nodes) that contain a question which separates the input data. The decision tree is traversed from root to leaf for each instance being classified, where the leaf node to which we arrive is the label for that instance. For example, a decision tree might be used to determine whether or not a living thing is a mammal, where each internal node is a question that helps to separate non-mammalian data instances from mammalian, and each leaf node is a label stating whether or not the living thing is a mammal. Each internal node should narrow down the final label as much as possible i.e., each question should give us the maximum information about the instance and should be arranged in the order that narrows it down as quickly as possible.

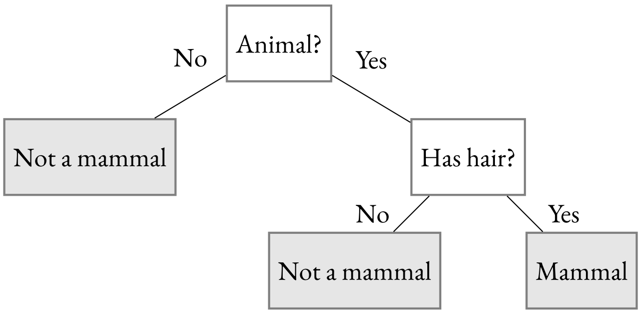

Figure 1: Simplified Decision Tree to Determine Whether a Creature is a Mammal

Decision trees have many advantages: they are visualisable by humans and aren't "black-box", they can model non-linear relationships easily, and they are robust to outliers. However, they have their disadvantages, including instability (small changes in the training data can significantly alter the tree structure) and in particular **overfitting**: when the algorithm fits too exactly to the training data, making it incapable of generalising to unseen data. An extreme example of overfitting would be if the example decision tree above started to ask far too specific questions, e.g. "Is it a dolphin", "Is it a human". While this would have excellent performance & accuracy on the test data, it would not work at all for an animal it hadn't encountered before.

Random forests work by combining many decision trees into a forest, thus improving accuracy & reducing overfitting by averaging multiple trees, reducing variance and leading to better generation. These decision trees are each generated using random, potentially overlapping subsets of the data training data. While the original random forest algorithm worked by taking the most popular label decided on by the set of trees[1], the scikit-learn `RandomForestClassifier` works by taking a probability estimate for each label from each tree and averaging these to find the best label[2].

In `RandomForestClassifier`, each tree is generated as follows:

1. A subset of the training data is randomly selected (hence the "Random" in the name of the algorithm). These subsets are selected "with replacement" which means that different trees can select the same samples: they are not removed from the pool once they are first selected. This results in unique, overlapping trees.

2. Starting with the root node, each node is *split* to partition the data. Instead of considering all features of the samples when choosing the split, a random subset of features is selected, promoting diversity across the trees. The optimal split is calculated using some metric such as Gini impurity or entropy to determine which split will provide the largest reduction in impurity.

3. This process is repeated at every node until no further splits can be made.

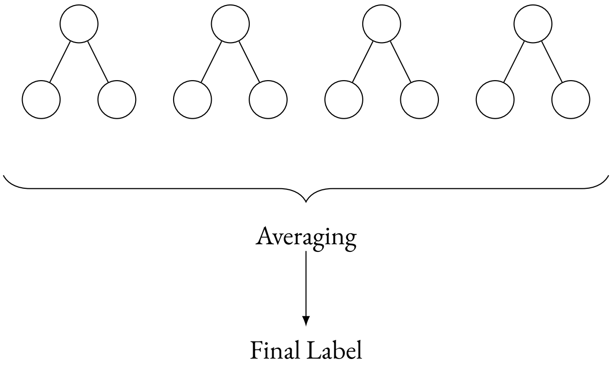

Figure 2: Random Forest Algorithm Example Diagram (with scikit-learn Averaging)

I chose the random forest classifier because it is resistant to overfitting, works well with complex & non-linear data like the wildfire data in question, handles both categorical & numerical features, and offers a wide variety of hyperparameters for tuning. It also has many benefits that are not particularly relevant to this assignment but are interesting nonetheless: it can handle both classification & regression tasks, can handle missing data, and can be parallelised for use with large datasets.

1.1.1 Hyperparameter 1: `n_estimators`

The hyperparameter `n_estimators` is an ***int*** with a default value of 100 which controls the number of decision trees (*estimators*) in the forest[4]. Increasing the number of trees in the forest generally improves the model's accuracy & stability, with diminishing marginal returns past a certain value, at the trade-off of increased computation & memory consumption. Each tree is independently trained, so there is a big trade-off between computational cost & performance. Using a lower number of estimators can result in underfitting, as there may not be a enough trees in the forest to capture the complexity of the data.

1.1.2 Hyperparameter 2: `max_depth`

The hyperparameter `max_depth` is an ***int*** with a default value of **`None`** which controls the maximum "depth" of each of the trees in the forest[4], where the "depth" of a tree refers to the longest path from the root node to a leaf node in said tree. With the default value of **`None`**, the trees will continue to grow until they cannot be split any further, meaning that each leaf node either only contains samples of the same class (i.e., in our case each leaf node is a definitive "yes" or "no") or contains a number of samples lower than the `min_samples_split` hyperparameter. The `min_samples_split` hyperparameter determines the minimum amount of samples needed for a node to be split; it has a default ***int*** value of 2 and therefore, since I am not tuning this hyperparameter for this assignment, it has no relevance as any leaf node that doesn't reach the minimum amount of samples to be split is a "pure" node by virtue of containing only one class.

High `max_depth` values allow for the trees to capture more complex patterns in the data, but can overfit the data, leading to poor testing accuracy. Bigger trees also naturally incur higher computational costs, requiring both more computation to create and more memory to store. Lower `max_depth` values result in simpler trees which can only focus on the most important features & patterns in the data, which in turn can reduce overfitting; however, low values run the risk of creating trees which are not big enough to capture the complexity of the data, and can lead to underfitting.

1.2 Algorithm 2: C-Support Vector Classification

C-Support Vector Classification is a supervised machine learning algorithm provided in scikit-learn as `sklearn.svm.SVC`, as it is a **Support Vector Machine (SVM)** based algorithm. It is capable of performing both binary & mutli-class classification, on both linearly and non-linearly separable data. While SVMs can also be used for regression as well as classification, I will only be considering their use in classification for this assignment.

SVM is a supervised learning algorithm that works by constructing a hyperplane in a multidimensional space that separates each data sample into a distinct class, i.e. it draws a "line" in N dimensions (where N is the number of features) that separates all the samples in one class from all the samples in the others. In mathematical terms, the data samples in this high-dimensional space are *vectors*, as the value of each feature for that sample determines the direction & magnitude of the object representing the sample in that space. The vectors closest to the boundary hyperplane are called **support vectors**: they are called this because they *support* the boundary between the two classes, as the algorithm is always trying to maximise the distance on either side of the boundary between the boundary and the nearest data samples. The further the decision boundary is from the outermost data points in each class, the more generalisable the classifier will be. This also reduces overfitting, as a decision boundary that is closely defined by specific data points will be highly sensitive to noise in the data & outliers, as a data point that is just a little bit away from another data point in its class could happen to be on the wrong side of the boundary and get misclassified.

However, if the data is not linearly separable, something called the **kernel trick** is employed. This is where data that is not linearly separable in its original N-dimensional space is mapped to a higher dimensional space where a hyperplane that delineates the two classes can be found. To find the similarity of two support vectors x & y without the kernel trick, we would have to calculate the dot product $\phi(x) \cdot \phi(y)$ where ϕ is some function that maps the data to a higher dimensional space, which is computationally expensive. The kernel trick avoids this by using some kernel function K which calculates the dot product in the higher dimensional space without having to first map the data to that higher dimensional space, so $K(x, y) = \phi(x) \cdot \phi(y)$.

1.2.1 Hyperparameter 1: `kernel`

The hyperparameter `kernel` specifies the kernel function to be used when employing the kernel trick. The possible values are:[3]

- `"linear"`: Assumes that the data is linearly separable, and does not transform it into a higher dimension, instead just directly computing the dot product $K(x, y) = x \cdot y$.
- `"poly"`: Calculates the polynomial of the features of data sample and considers the polynomial relationships as well as the linear relationships between the data samples. The formula for this kernel is $K(x, y) = (x \cdot y + c)^d$ where c is a constant & d is the degree of the polynomial.
- `"rbf"`: The Radial Basis Function (also called the Gaussian kernel) calculates the similarity between the data samples in infinite dimensions. The formula for this kernel is $K(x, y) = \exp(-\gamma \cdot ||x - y||^2)$ where γ determines the *width* of the kernel (how far the influence of a data sample reaches). This is the default kernel for the `sklearn.svm.SVC` class.
- `"sigmoid"`: Calculates the similarity between the data samples using the hyperbolic tangent function, which creates an S-shaped curve called a *sigmoid* and determines which side of the sigmoid a data target is on to classify it. The formula for this function is $K(x, y) = \tanh(\gamma \cdot x \cdot y + r)$ where γ controls the influence of a data sample as before and r controls a bias that shifts the boundary defined by the sigmoid function to be more or less lenient.
- `"precomputed"`: Instead of specifying a kernel function to use on the data, one can specify a matrix that contains the pre-computed kernel function output of your choosing, which can save calculations being performed twice.

1.2.2 Hyperparameter 2: `C`

The hyperparameter `C` is called the **regularisation parameter** because it controls how misclassified samples are penalised, i.e. controlling how "regularised" or smooth the boundary hyperplane is. It is a ***`float`*** with a default value of 1.0. A higher value for `C` will result in misclassifications being heavily penalised, which in the extreme case will results in the model prioritising correctly classifying each data sample over having a wider margin, which can make the model less generalisable and cause overfitting. On the other hand, a lower value for `C` will result in a model that has a greater acceptance of misclassifications in return for a wider margin, which can improve generalisability but reduce precision.

2 Model Training & Evaluation

I chose to write the Python code to train & evaluate the models in a Jupyter notebook in Google Colab: I used a Jupyter Notebook just for the easy rendering of plots within the IDE and I used Google Colab to have access to faster hardware than my own. For the hyperparameter tuning of both algorithms, I created an array of values for each hyperparameter that were iterated over in a nested **`for`** loop, and created a heatmap of the accuracy results.

2.1 Algorithm 1: Random Forest

Since the Random Forest algorithm can handle categorical features, the dataset had no missing data, and random forests don't require feature scaling, I did not perform any pre-processing for this algorithm. The first thing I did was train a `RandomForestClassifier` with the default parameters as a control variable to which I could compare my hyperparameter tuning. The only thing I changed from default for this was that I set the random seed `random_state = 0` so that I would get the same reproducible output for the same code, regardless of how many times it was ran as otherwise it would be exceedingly difficult to draw meaningful comparison between the subsequent runs of the algorithm. While this is a parameter that is supplied to the model, it is not a *hyperparameter* as it does not control the behaviour of the model in terms of learning or performance; it cannot be tuned, and there is no optimal value.

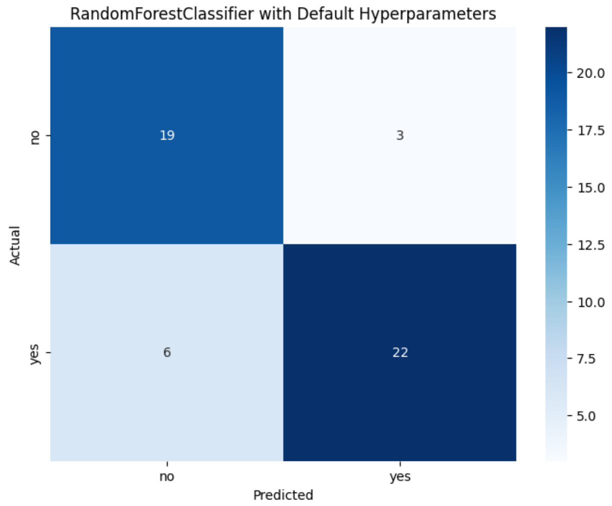

Figure 3: Confusion Matrix

```
Training Accuracy: 1.0000
Classification Report of Testing Results:
              precision    recall  f1-score   support

          no       1.00      1.00      1.00        75
         yes       1.00      1.00      1.00        79

    accuracy                           1.00       154
   macro avg       1.00      1.00      1.00       154
weighted avg       1.00      1.00      1.00       154
```

Listing 1: Training Accuracy & Classification Report

```
Testing Accuracy: 0.8200
Classification Report of Testing Results:
              precision    recall  f1-score   support

          no       0.76      0.86      0.81        22
         yes       0.88      0.79      0.83        28

    accuracy                           0.82        50
   macro avg       0.82      0.82      0.82        50
weighted avg       0.83      0.82      0.82        50
```

Listing 2: Testing Accuracy & Classification Report

When I was tuning the hyperparameter values, I noticed that the algorithm seemed to be performing very well with low values of `n_estimators` & `max_depth` so the array of values tested starts with very small increments between the values, and then grows into quite large increments at the end.

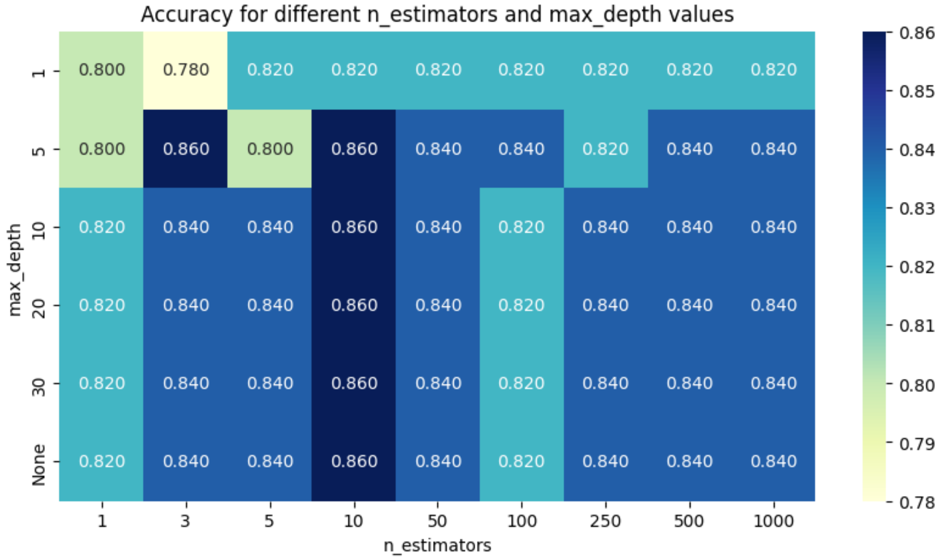

Figure 4: Accuracy Heatmap

I obtained a training accuracy of 100% using the default hyperparameter values: this indicates overfitting, and we saw that the accuracy did drop on the test data to 82%. The most likely cause of this overfitting is that `max_depth` defaults to **`None`**, which can create very complex trees that perform well on the training data, but aren't generalisable to unseen data.

The hyperparameter tuning results were very surprising to me. The accuracy results obtained were not that unusual: we obtained a +4% increase for our maximum accuracy value, going from 82% to 86%, and the lowest accuracy we obtained was 80%, indicating that the model's performance was stable across a range of hyperparameters. It was surprising that 10 proved to be the optimal value of `n_estimators` for this dataset, as this is significantly lower than the default value of 100, and a higher number of estimators typically results in improved accuracy & stability. My hypothesis is that the low number of trees in the forest helped to prevent overfitting caused by the relatively high `max_depth`. We can see from the data that for a low `max_depth = 1`, higher `n_estimators` yield better accuracy, which supports my idea that the low number of trees is helping to prevent overfitting by the deep, complex trees. It looks like that a relatively small `max_depth` of 5 through to **`None`** was suitable when paired with `n_estimators = 10`, perhaps because the dataset was sufficiently small & non-complex that a shallower tree could still capture its complexity.

2.2 Algorithm 2: C-Support Vector Classification

Since SVMs don't support categorical data, I first encoded the `fire` column to have numerical values using `OneHotEncoder`. I didn't standardise the data as a pre-processing step as it appeared to be beyond the scope of this assignment (and I'm pushing the page limit on this assignment as is without including extra bits), but it is often important to do so, as SVMs are very sensitive to scale in the sample data. As in §?? ??, I first began with the default values as a control to which I could compare my tuning.

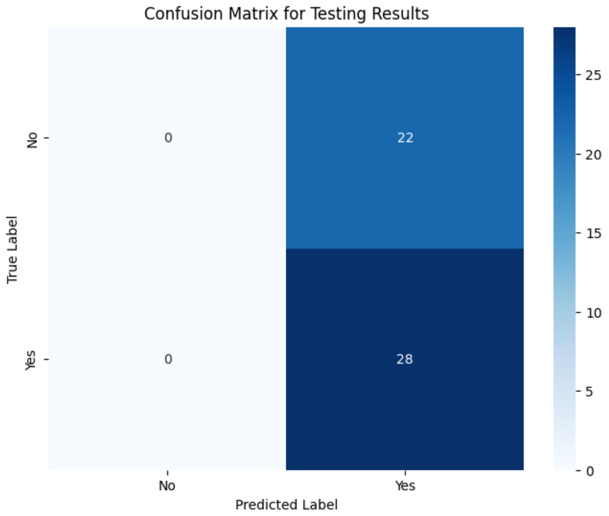

Figure 5: Confusion Matrix

```
Training Accuracy: 0.5130
Classification Report of Training Results:
              precision    recall  f1-score   support

         0.0       0.00      0.00      0.00        75
         1.0       0.51      1.00      0.68        79

    accuracy                           0.51       154
   macro avg       0.26      0.50      0.34       154
weighted avg       0.26      0.51      0.35       154
```

Listing 3: Training Accuracy & Classification Report

```
Testing Accuracy: 0.5600
Classification Report of Testing Results:
              precision    recall  f1-score   support

         0.0       0.00      0.00      0.00        22
         1.0       0.56      1.00      0.72        28

    accuracy                           0.56        50
   macro avg       0.28      0.50      0.36        50
weighted avg       0.31      0.56      0.40        50
```

Listing 4: Testing Accuracy & Classification Report

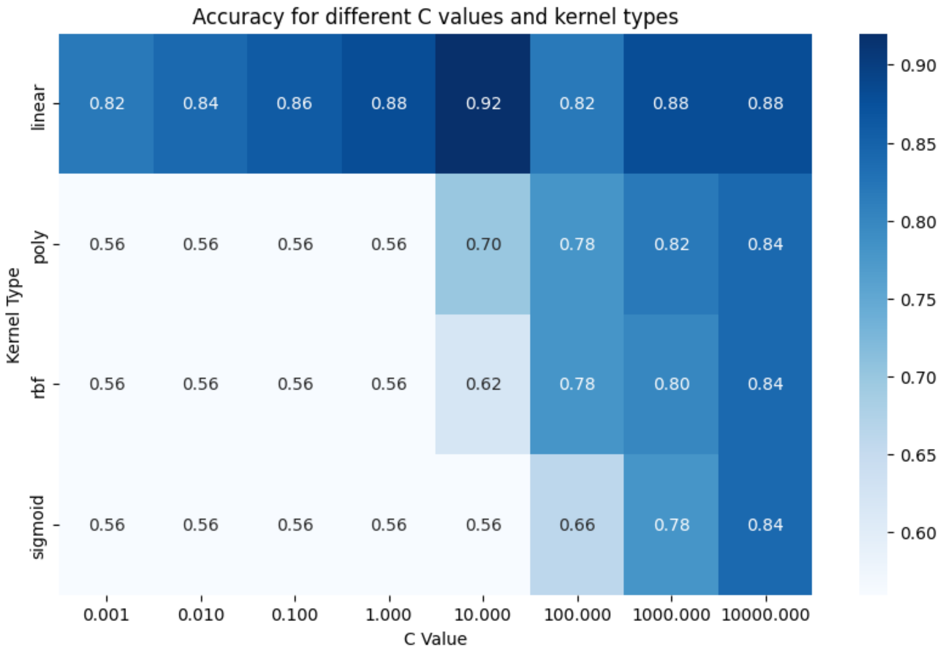

Figure 6: Accuracy Heatmap

We can see from the heatmap that `kernel = "linear"` is far superior to any other kernel function. This indicates that our data is linearly separable. However, it's worth noting that we are only looking at tuning two hyperparameters here: some of these kernels have additional hyperparameters, and likely could've performed better if they were tuned appropriately. Looking at the row corresponding to the linear kernel function in the heatmap, we can also see a nice demonstration of the effect of different `C` values: the accuracy of the model steadily increases as we increase the penalty for misclassifications, but after a certain point our accuracy starts to decrease. This is likely because the very high `C` values (100 and above) were causing a very narrow margin when the boundary was being drawn, causing overfitting, as the boundary was not generalisable and performed poorly on the unseen data. The relatively high optimal value for `C` indicates that the test data is quite similar to the training data, as a narrow margin can be maintained without sacrificing accuracy in the test. We can see that placing a weak penalty for misclassification naturally reduces accuracy. What's interesting about this data is that the very high `C` values seem to be able to force an accurate boundary even for kernel functions that are unsuitable for this type of data.

The confusion matrix for the model with the default hyperparameters is quite interesting: it seems that this model only ever produced the label "yes" and never "no", resulting in an "accuracy" of 51%: if you only ever guess "yes" and the options are an equally distributed set of "yes" and "no", you'll be right about half the time, even if there's no logic to your guess. Based off the hyperparameter tuning, we can see that this is because the default hyperparameter values for this model are very different to the optimal ones for our data. Firstly, we can see that from the heatmap that the RBF kernel function performed quite poorly on our data: it was the second worst kernel function in terms of accuracy. However, this is only half the story, as we can see that with a suitably high `C` value, we can obtain a reasonably high accuracy even with RBF. Clearly, the default value of `C = 1.0` is far too low for our data (the optimal for RBF was 10,000). This is may be in part due to the fact that I did not scale the data before training & testing, and the high `C` value is helping to compensate for this.

3 Conclusion

3.0.1 Key Findings

The key findings for each algorithm are as follows:

- Random Forest:
  - Accuracy with default hyperparameter values: 82%.
  - Accuracy with hyperparameter tuning: 86%.
  - The model demonstrated overfitting with a training accuracy of 100%, dropping to 82% on the test data, indicating that the default values overfit the data.
  - Hyperparameter tuning revealed that a lower number of decision trees (`n_estimators = 10`) and a relatively small maximum depth (`max_depth = 5`) achieved the best balance between performance and computational efficiency, with an optimal test accuracy of 86%.
  - The model's performance remained stable across different hyperparameters, showing that it is robust to small changes.
- C-Support Vector Classification:
  - Accuracy with default hyperparameter values: 56%.
  - Accuracy with hyperparameter tuning: 92%.
  - The model initially performed poorly with the default hyperparameters, yielding a testing accuracy of 56%.
  - Tuning the C parameter (`C = 100`) and using the linear kernel significantly improved performance, suggesting that the data is linearly separable.
  - The linear kernel consistently outperformed other kernel options, demonstrating its suitability for the dataset.

Random Forest performed the best without any hyperparameter tuning, and its accuracy only slightly increased with tuning. C-Support Vector Classification performed the best with hyperparameter tuning, achieving a quite respectable 92% accuracy, despite having absolutely terrible performance with the default hyperparameter values.

3.0.2 Recommended Hyperparameter Values Based Off Results

Recommended hyperparameter values for Random Forest:

- `n_estimators = 10`: While an accuracy of 86% could be obtained with `n_estimators = 3`, and lower values are preferable due to the reduced computation required, I recommend the value 10 as it consistently produced high-accuracy predictions. Subsequent tests of the model with different random seeds indicated that the isolated instance of `n_estimators = 3` producing optimal accuracy was most likely a fluke caused by the random nature of the algorithm.

- `max_depth = 5`: Peak accuracy was reached with a maximum depth of 5, and so the lower value is preferable as it reduces the computation needed.

Recommended hyperparameter values for C-Support Vector Classification:

- `kernel = "linear"`: The linear kernel far outperformed any other kernel function, indicating that this data is linearly separable. This is also the most computationally efficient kernel function, as it does not require translation into a higher dimension.

- `C = 100`: The strict penalty for misclassification helped to create a highly accurate decision boundary that was still generalisable to our test data due to its similarity to the training data.

References

[1] Leo Breiman. “Random Forests”. In: *Machine Learning* 45 (2001).

[2] scikit-learn Documentation. *Ensembles: Gradient boosting, random forests, bagging, voting, stacking*. URL: `https://scikit-learn.org/stable/modules/ensemble.html` Accessed on: 2024-10-06.

[3] scikit-learn Documentation. *Plot classification boundaries with different SVM Kernels*. URL: `https://scikit-learn.org/stable/auto_examples/svm/plot_svm_kernels.html#sphx-glr-auto-examples-svm-plot-svm-kernels-py` Accessed on: 2024-10-06.

[4] scikit-learn Documentation. *`RandomForestClassifier` API Reference*. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html` Accessed on: 2024-10-06.

[5] scikit-learn Documentation. *Support Vector Machines*. URL: `https://scikit-learn.org/stable/modules/svm.html` Accessed on: 2024-10-06.

[6] scikit-learn Documentation. *`SVC` API Reference*. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC` Accessed on: 2024-10-06.

[7] Rohith Gandhi. *Support Vector Machine — Introduction to Machine Learning Algorithms*. URL: `https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47` Accessed on: 2024-10-06.

[8] GeeksforGeeks. *How to Choose the Best Kernel Function for SVMs*. URL: `https://www.geeksforgeeks.org/how-to-choose-the-best-kernel-function-for-svms/` Accessed on: 2024-10-06.

[9] IBM. *What are support vector machines (SVMs)?* URL: `https://www.ibm.com/topics/support-vector-machine` Accessed on: 2024-10-06.

[10] IBM. *What is random forest?* URL: `https://www.ibm.com/topics/random-forest` Accessed on: 2024-10-06.

[11] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

All diagrams, figures, & graphs are original and created by me.